Application Development using Oracle8i PL/SQL
Introduction

PL/SQL is an easy-to-use, high-performance procedural language extension of SQL. Many Fortune 500 companies have successfully used the PL/SQL platform to build mission-critical applications. In one of the customer surveys done in 1998, it was voted as the most important and the most satisfying feature from Oracle. It was also voted as being very important for businesses in the next several years.

Release 8.1.5 of Oracle8i PL/SQL is full of rich functionality geared towards building even faster and even better applications much more easily. Moreover, if you simply upgrade your database to Oracle8i, your existing PL/SQL applications will automatically run faster without your having to change a single line of code. This presentation presents the features that make development with Oracle8i PL/SQL easier and better, enhancements in the areas of development and analysis tools, support for interoperability with Java on the server, and enhancements that help build powerful database-centric web applications. It also briefly describes improvements in the area of performance; for more details about performance, refer to the Oracle OpenWorld1999 presentation entitled “Techniques for Developing Faster PL/SQL Programs”. This presentation also briefly discusses features which ease the development of web applications; for more details about this, refer to the presentation entitled “Web Enabling PL/SQL Based Database Applications”.

Page 4
Powerful New Features for Application Development

- Autonomous Transactions
- Dynamic SQL
- Invoker’s Rights
- Querying local collection variables
- UROWID type
- SQL99 Core Compliance features

Oracle8i provides new features, such as autonomous transactions, native dynamic SQL, invoker-rights procedures, and so on that can streamline the development process, allow for code reuse, and provide developers with more flexible ways to include unique functionality in business solutions.
Autonomous Transactions

An independent transaction started by another transaction

A transaction is a series of statements that does a logical unit of work. Often, one transaction starts another. Transactional semantics require that an entire logical unit of work completes or fails as an integrated unit.

In some applications, a transaction may need to operate outside the scope of the transaction that started it. In other words, an independent transaction may need to be created from within an existing transaction. You may want to commit or roll back some changes to a table independently of the final outcome of a primary transaction.

For example, in a stock purchase transaction, you may want to commit a customer's information regardless of whether the overall stock purchase actually goes through. Or, while running that same transaction, you may want to log error messages to a debug table even if the overall transaction rolls back.

Oracle8i has introduced the concept of autonomous transactions, to perform such tasks.

An autonomous transaction (AT) is an independent transaction started by another transaction, the main transaction (MT). With an AT, you can suspend the main transaction, do SQL operations, commit, or roll back those operations, then resume the main transaction.
Autonomous Transactions

- **Benefits:**
  - Building reusable components
  - Improved logging
- **Add** `PRAGMA AUTONOMOUS_TRANSACTION` in:
  - Schema-level anonymous blocks
  - Local, stand-alone, and packaged procedures and functions
  - Methods of a SQL object type
  - Database triggers

**Autonomous Transactions (Description, Benefits, Syntax):**

Prior to Oracle8i, if you wanted your PL/SQL program to log a message in the midst of a transaction without having to worry about whether the transaction were to commit or rollback, you could do so by creating a separate logging facility in C that you would call using the DBMS_PIPE package. However, this approach results in clumsy, awkward, and unintuitive code. Oracle8i allows you to perform this type of logging operation much more simply by using autonomous transactions.

A more important benefit is that autonomous transactions help you build modular, reusable software components. For example, stored procedures can start and finish autonomous transactions on their own. A calling application need not know about a procedure’s autonomous operations, and the procedure need not know about the application’s transaction context. That makes autonomous transactions less error-prone than regular transactions and much easier to use.

To define autonomous transactions, specify the `AUTONOMOUS_TRANSACTION` pragma anywhere in the declarative section of any of the following:

1. schema-level (not nested) anonymous blocks,
2. local, stand-alone, and packaged functions and procedures,
3. methods of a SQL object type,
4. database triggers.
Autonomous Transactions - Semantics

When an autonomous transaction block (i.e., a block with pragma AUTONOMOUS) is entered, the main (parent) transaction is suspended and a new, independent, fully functional, autonomous transaction is started with the execution of the first DML or query operation in the autonomous transaction block. You can perform as many SQL operations as you like, and commit and rollback these operations, at which point the autonomous transaction completes. The next SQL operation in this autonomous transaction block results in the creation of a new autonomous transaction, so that you can have a series of autonomous transactions in the same block. Note that the main transaction resumes only when you exit the autonomous transaction block. If an autonomous transaction is not committed or rolled back, an error will be reported.

Autonomous transactions do not share resources. Consequently, if a main transaction locks a table for update and an autonomous transaction attempts to lock the same table for update, a deadlock will be reported, since the main (suspended) transaction that’s holding the lock is waiting for the autonomous transaction block to exit while the autonomous transaction is waiting for the main transaction to release the lock.

Changes committed in an autonomous transaction are visible globally. This means that the parent transaction (when it resumes) and other active transactions will be able to see these changes, provided their isolation levels are set to “read committed”, the default setting. However, if a transaction’s isolation level is set to “serializable” or “read only”, it can see only those changes committed when it began, plus, in the case of serializable transactions, those changes made by the transaction itself.
Autonomous Transactions

- Example:

```sql
CREATE PROCEDURE debug_log1 (msg varchar2) AS
BEGIN
    INSERT INTO debugtable VALUES (msg);
    -- can’t afford to COMMIT here
END debug_log1;
/

CREATE PROCEDURE debug_log2 (msg varchar2) AS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO debugtable VALUES (msg);
    COMMIT; -- need this
END;
/
```

Autonomous Transactions - Semantics (continued):

Autonomous transactions can see changes committed by other transactions (including the parent transaction). However, uncommitted changes made by other transactions (including the parent transaction) are not visible to autonomous transactions. If an autonomous transaction's isolation level is set to “serializable” or “read only”, then it can see only those changes committed at the start of the transaction, plus, in the case of serializable transactions, those changes that are committed by this autonomous transaction. In the case of “read committed” autonomous transactions, all changes committed by other transactions are visible to the autonomous transaction.

Autonomous Transactions (Example):

In the example on this slide, we have two procedures, “debug_log1” and “debug_log2”, each of which accepts a message as a parameter and logs this message by inserting it into the table “debugtable”.

Since both procedures are meant for only logging the message, we do not want to change the transaction state of the caller. In the case of “debug_log1”, which is a regular procedure, we cannot commit the change because doing so would result in changing the transaction state of the main transaction (of the caller). In the case of “debug_log2”, we can safely commit the change since “debug_log2” is an autonomous transaction block, which means that changes committed here only affect the autonomous transaction without changing the state of the main transaction.

The next slide shows how these procedures are used to log a message to the “debugtable” table.
Autonomous Transactions

Example (continued)

BEGIN
    UPDATE emp SET sal = 1000000 WHERE ename = 'KING';
    debug_log1('tried to update KING');
    UPDATE emp SET sal = 1000000 WHERE ename = 'FORD';
    debug_log2('tried to update FORD');
    ROLLBACK;
    -- at this point we lose all record of the
    -- attempt to change KING’s salary; but we
    -- do retain a record of the attempt to
    -- change FORD’s salary
END;

Autonomous Transactions (Example - continued):

In this block, we attempt to change both KING’s as well as FORD’s salary to $1,000,000. However, we subsequently roll back the changes.

In the case of the attempt to change KING’s salary, we attempt to log this activity by calling “debug_log1”. However, since the logging in “debug_log1” happens in the same transaction as the main transaction, there is no record left of the log since the main transaction is rolled back.

In the case of the attempt to change FORD’s salary, we attempt to log this activity by calling “debug_log2”, which is an autonomous transaction block. Since the logging in “debug_log2” happens in the autonomous transaction which is committed before “debug_log2” exits, this record continues to exist even after the main transaction is rolled back.
Native Dynamic SQL

- **Description:**
  - Ability to dynamically build and submit SQL statements from PL/SQL

- **Benefits:**
  - Used in applications that allow users to choose query search criteria or optimizer hints at run time

- **Pre-Oracle8i PL/SQL solution:**
  - Use DBMS_SQL

Dynamic SQL

Dynamic SQL refers to the ability to build and submit SQL statements (including DML, DDL, transaction control, session control, and anonymous block statements) at run time.

Dynamic SQL is useful when not all of the necessary information is available at compilation time - for example, when using a database query tool that allows you to choose query search criteria or optimizer hints at run time. Dynamic SQL lets you do things like create a procedure that operates on a table whose name is unknown until run time, or accept user input that defines the SQL statement to execute.

Prior to Oracle8i, PL/SQL developers could include dynamic SQL in applications by using the Oracle-supplied DBMS_SQL package. However, performing simple operations using DBMS_SQL involves a fair amount of coding. In addition, because DBMS_SQL is based on a procedural API, it incurs high procedure-call and data-copy overhead.
Native Dynamic SQL

Benefits of native dynamic SQL over DBMS_SQL:

- Highly intuitive syntax
- A lot less code
- Much easier to write & maintain code
- Much faster (30% to 400%)

Oracle8i PL/SQL supports dynamic SQL natively in the language.

In contrast with DBMS_SQL, the highly intuitive syntax for native dynamic SQL makes it much easier to write and maintain your code. In addition, native dynamic SQL code is considerably more compact. Furthermore, native dynamic SQL bundles the statement preparation, binding, and execution steps into a single operation, thereby improving performance. Internal Oracle benchmark tests using native dynamic SQL show a 30 to 400 percent performance improvement over DBMS_SQL.
Native Dynamic SQL (EXECUTE IMMEDIATE statement)

The EXECUTE IMMEDIATE statement in PL/SQL supports native dynamic SQL. It lets you prepare, execute, and deallocate a SQL statement dynamically at run time.

The syntax for this is fairly simple, as shown on this slide. Note the dynamic string in the syntax. You can fetch this string into a list of define variables or into a record variable using the optional INTO clause. You can bind arguments through the optional USING clause.

One limitation in 8.1.5 is that you cannot specify a dynamic number of bind variables in the USING clause. There is currently no way to dynamically describe the statement to find out how many binds there are, and what their exact types are, etc. For some limited scenarios, however, there is a simple workaround if you know what the type of the bind arguments are. The workaround involves using an IF statement to conditionally create your dynamic SQL statement or query and then using another IF statement later to conditionally execute the SQL statement with a separate list of bind arguments in the USING clause for each condition. This workaround is useful when you do not have a large variation on the list of bind variables.
Native Dynamic SQL (Example)

In this example, the INSERT statement is built at run time in the “stmt_str” string variable using values passed in as arguments to the “insert_into_table” procedure. The SQL statement held in “stmt_str” is then executed via the EXECUTE IMMEDIATE statement. The bind variables “:deptno”, “:dname”, and “:loc” are bound to the arguments which, in this case, are the parameters “deptnumber”, “deptname”, and “location”.

```
PROCEDURE insert_into_table (  
    table_name varchar2,  
    deptnumber number,  
    deptname varchar2,  
    location varchar2)  
IS  
    stmt_str varchar2(200);  
BEGIN  
    stmt_str := 'insert into ' || table_name ||  
                ' values (:deptno,:dname,:loc)';  
    EXECUTE IMMEDIATE stmt_str  
        USING deptnumber, deptname, location;  
END;
```
Native Dynamic SQL (Example of Native Dynamic SQL vs. DBMS_SQL)

The example on the previous slide is shown above in conjunction with code that was written to perform the same operation using DBMS_SQL. The struck out lines are the lines you would have needed to use DBMS_SQL. This example clearly demonstrates how much easier it is to write and read native dynamic SQL code compared with calls to DBMS_SQL.
Invoker’s Rights

- **Description:**
  - Ability to execute procedure with privileges of user calling procedure

- **Benefits:**
  - Scalability (disk and memory): write and install a single copy of code that can be invoked by multiple users to manipulate objects in different schemas
  - Security at finer granularity
  - Installation time vastly reduced

**Invoker’s Rights (Description, Benefits):**

By default, a stored procedure or method executes with the privileges of, and in the name-resolution context of, its owner (definer). Such procedures, known as “definer’s-rights procedures”, were the only option in previous releases.

In the invoker-rights model, introduced in Oracle8i, programs are executed with the privileges of the calling user. An invoker-rights program inherits the privileges and name-resolution context of the user calling the program (the invoker) for the execution of all SQL statements in the program. Implementing invoker’s rights procedures substantially enhances code reuse by letting you centralize data administration.

For example, you can install, in a central location, a generic software procedure that executes data definition (DDL) or data manipulation (DML) statements on behalf of several users who need to access data in different schema. Since an invoker’s-rights procedure automatically inherits its invoker’s name-resolution and privilege contexts for all SQL statements in the procedure, multiple users can invoke the centrally located copy to access private data owned by the different users.
Invoker’s Rights

The syntax of the following statements has been altered so that you can use the INVOKER_RIGHTS clause:

- CREATE FUNCTION
- CREATE PROCEDURE
- CREATE PACKAGE
- CREATE TYPE

In these statements, set AUTHID to CURRENT_USER to specify invoker’s rights. Set AUTHID to DEFINER (the default), to specify definer’s rights.

Name Resolution

For a definer rights procedure, all external references are resolved in the definer’s schema. For an invoker rights procedure, the resolution of external references depends on the kind of statement they appear in:

- Names used in queries, DML statements, dynamic SQL, and DBMS_SQL are resolved in the invoker’s schema.
- All other statements, such as calls to packages, functions, and procedures, are resolved in the definer’s schema.
Invoker’s Rights (Example - figure):

In this rather simple example, let’s suppose that we have two users, Sarah and Grace. Each of them wishes to determine how many employees report to them using information in a common schema SCOTT and insert that information into a private EMPCOUNT table. This effect may be achieved by writing a procedure to do this and installing a separate copy of the procedure in each of the user’s schemas. This could, however, have pretty negative consequences when you have several people, each of whom is installing a separate copy of the code. Firstly, a large amount of disk space is necessary. Secondly, a large amount of SGA memory is used at run time. Thirdly, the time it would take to install separate copies of the code in each of the separate schemas can become a major issue.

With Oracle8i, SCOTT can now install, in his schema, a single copy of this code as an invoker’s rights program, “name_count”, to provide the same functionality. As will become evident from the code on the next slide, this program has an unqualified reference to the name EMPCOUNT; separate copies of this table are owned by SARAH and GRACE. The program also has a qualified reference to the name SCOTT.EMP, which is a table that SCOTT owns. Procedure “name_count” does a SELECT from SCOTT.EMP and an INSERT into EMPCOUNT. In order to compile procedure “name_count” without error, SCOTT needs to own a dummy table called EMPCOUNT which has the same structure as Grace’s and Sarah’s EMPCOUNT. At run time, the reference to EMPCOUNT will get bound to the actual tables in the caller’s (Grace or Sarah) schema. If, at run time, the actual table does not exist, an error is reported. On the other hand, the reference to SCOTT.EMP will resolve to the table EMP in SCOTT’s schema, since this is a qualified name.
Invoker’s Rights

- **Example (code):**

```sql
CREATE OR REPLACE PROCEDURE name_count
AUTHID CURRENT_USER AS
  mycount number;
BEGIN
  -- caller needs SELECT priv on SCOTT.EMP
  SELECT count(*) INTO mycount
  FROM scott.emp e, scott.emp m
  WHERE e.mgr = m.empno AND m.ename = user;
  -- caller needs UPDATE privilege on empcount
  INSERT INTO empcount VALUES (mycount);
END;
```

Invoker’s Rights (Example - code):

This slide contains the code for the example on the previous slide.

Note, again, that the reference to SCOTT.EMP is a qualified reference (qualified by the user name SCOTT). When the SQL statement is executed at run time, the reference to SCOTT.EMP is bound to the table EMP in SCOTT’s schema. In order to be able to select from SCOTT.EMP, the invoker (in this case, either Sarah or Grace), will need to have been granted SELECT privilege on SCOTT.EMP.

Note also that Sarah and Grace will need to have been granted EXECUTE privilege on procedure “name_count”.
Other Application Development Features

- Querying local collection variables
  - 8.0; ability to specify a local collection variable of a nested table type or a VARRAY type in the FROM clause of a SELECT stmt.
- UROWID type
  - Universal rowid that can store physical, logical, and non-Oracle rowids
- SQL99 Core Compliance features
  - TRIM, CAST, new date/time types
Transparent Ease-of-Use

- Easier callability from SQL
- Larger package bodies
- Improved diagnostics for ORA-6502 error

**Transparent Ease-of-Use**

Oracle8i PL/SQL has been enhanced significantly to make it easier to use. You can now call PL/SQL functions much more easily from SQL, create much larger package bodies, and take advantage of the much improved ORA-6502 error diagnostics.
Easier Callability from SQL

- Pragma RESTRICT_REFERENCES not needed
- Description:
  - Relaxed purity rules
  - Compilation time checks not needed
  - Run time checks performed instead

Easier Callability from SQL (Pragma RESTRICT_REFERENCES not needed - Description):

Prior to Oracle8i, the PL/SQL compiler performed various checks at compilation time to determine if a function was callable from SQL, whether a query or DML statement calling this function was parallelizable, and whether a query could be rewritten by the optimizer. These checks were controlled through pragma RESTRICT_REFERENCES and various rules. However, the rules were too strict and hard to understand.

In Oracle8i, the rules are vastly simplified so that the pragma is no longer required, although you may still use it. Instead, the following checks are performed at run time:

1. A function called from a query or a parallelized DML statement must not execute any DML or in any other way modify the database.
2. A function called from a query or DML statement must not end the current transaction, create or rollback to a savepoint, or ALTER the system or session.
3. A function called from a DML statement must not read or modify the particular table being modified by the DML statement.

The main benefits of the relaxed rules are:

1. Developers no longer need to worry about using the pragma correctly.
2. The relaxed rules provide more flexibility so that you can now safely call more functions from SQL where you couldn’t do so before.
3. The rules for SQL calling PL/SQL, C, and Java procedures are now consistent. Note that it is not possible to do compilation time checks to guarantee RNPS, WNPS, WNDS & RNDS for procedures written in C and Java.
Easier Callability from SQL (Pragma RESTRICT_REFERENCES not needed - Benefits):

The main benefits of the relaxed rules are:

1. Developers no longer need to worry about using the pragma correctly.
2. The relaxed rules provide more flexibility so that you can now safely call more functions from SQL where you couldn’t do so before.
3. The rules for SQL calling PL/SQL, C, and Java procedures are now consistent. Note that it is not possible to do compilation time checks to guarantee RNPS, WNPS, WNDS & RNDS for procedures written in C and Java.
Easier Callability from SQL (TRUST Option - Description, Benefits):

Oracle has extended the RESTRICT_REFERENCES pragma to add a new TRUST option. The TRUST keyword is an indication to the compiler that it should not verify that a given procedure or function body satisfies the pragma that is associated with the specification. Instead, the compiler “trusts” that the body indeed satisfies the pragma.

If you choose or need to use the pragma with C or Java programs, the TRUST option is necessary if you want to call these C or Java programs from SQL or PL/SQL. This is also useful if you want to write new PL/SQL subprograms that do not have the pragma and be able to call them from PL/SQL programs that do have the pragma.

An example:

Suppose procedure A1 calls A2 which calls A3 and that all of these have the pragma. Suppose also that procedure B1 calls B2 which calls B3, all of which do NOT have the pragma. Suppose now that you want A3 to call B1, but you have been unable to do so because B1 does not have the pragma.

You can do one of two things:

1. If you have the right to modify the source code for B1, simply add the pragma with the TRUST option, thus telling the compiler to “trust” that all calls to B1 (which in turn calls B2, etc…) satisfy the pragma.

2. If you don’t have the right to modify the source code for B1, create a wrapper procedure B0 which has the same signature as B1 and which calls B1. Then add the pragma to B0 with the TRUST option, thus telling the compiler to “trust” that all calls to B0 (which in turn calls B1, which calls B2, etc…) satisfy the pragma.
Easier Callability from SQL (DETERMINISTIC, PARALLEL_ENABLE - Description):

Oracle8i provides you with additional functionality to make it easier to call PL/SQL, C and Java procedures from SQL through the DETERMINISTIC and PARALLEL_ENABLE qualifiers on the function specification.

The DETERMINISTIC qualifier is a hint to the compiler that the function will ALWAYS return the same result for a given set of input argument values. This qualifier is mandatory for all functions used in function-based indexes, which is a new feature in Oracle8i which allows you to build indexes on functions and not just columns. This qualifier is also used by the optimizer to decide what plan to generate for materialized views with ENABLE QUERY REWRITE, specifically, whether the SQL engine may safely choose to use a cached copy of the returned result.

The PARALLEL_ENABLE qualifier is a hint to the compiler that it is safe for a function to be called in a parallel query or parallel DML statement. Prior to Oracle8i, the WNPS & RNPS options of the RESTRICT_REFERENCES pragma were used to indicate this.
Easier Callability from SQL (DETERMINISTIC, PARALLEL_ENABLE - Syntax):

The syntax for specifying the DETERMINISTIC and PARALLEL_ENABLE queries is fairly simple as shown on the slide above. The keyword must be specified just before the IS or AS keyword.

Note that the specified syntax applies to call-specifications for Java and C as well.
Prior to Oracle8i, large packages often failed to compile due to an internal compiler limit. As a result, developers often had to decompose packages into smaller units.

In Oracle8i, Oracle has substantially increased the compiler limit. Although it’s not easy to correlate the new internal limit with a specific number of lines of code, application developers will be able to include up to about 6 million lines of code in a single package body or type body, provided other limits are not reached. Note that this new limit applies only to package bodies and type bodies. Note also that you might start running into other internal limits you had not run into before.

The biggest benefit here is that you don’t have to be frustrated by the “PLS-123: Program too large” error and don’t have to restructure your code in a way that’s not intuitive.
Improved ORA-6502 Error

- The PL/SQL run time engine now gives you more information with ORA-6502 errors
- Examples:

```sql
declare x varchar2(2); begin x := 'abc'; end;
/
ORA-6502: PL/SQL: numeric or value error:
character string buffer too small
```

```sql
declare x number; begin x := 'abc'; end;
/
ORA-6502: PL/SQL: numeric or value error:
character to number conversion error
```

Improved ORA-6502 Error:

Prior to Oracle8i, the “ORA-6502: PL/SQL numeric or value error” provided no information about the cause of the error. With Oracle8i, this error message provides more specification information. For example, if you try to assign a large string to a string variable whose length is smaller than that of the large string, you will now see the additional text: “character string buffer too small” as part of the error message. If you try to assign a string to a number variable and the implicit conversion results in an error, as shown in the example on the slide, you will now see the additional text “character to number conversion error”.

Page 28
Improved Performance

- Bulk Binds
- NOCOPY parameter passing hint

Improved Performance
Oracle8i PL/SQL offers new performance-enhancing language features, such as bulk binds and the NOCOPY parameter-passing modifier, that you can take advantage of in your applications. In this section, we briefly introduce these features. For more details relating to performance, refer to the presentation “Techniques for Developing Faster PL/SQL Applications”.
The Oracle server has three execution engines: one each for PL/SQL, SQL, and Java. The above diagram shows the PL/SQL and SQL engines. When running PL/SQL blocks and subprograms, the PL/SQL engine runs procedural statements but sends the SQL statements to the SQL engine, which parses and executes the SQL statement and, in some cases, returns data to the PL/SQL engine. During execution, every SQL statement causes a context switch between the two engines, which results in a performance penalty.

Performance can be improved substantially by minimizing the number of context switches required to run a particular block or subprogram. When a SQL statement runs inside a loop that uses collection elements as bind variables, the large number of context switches required by the block can cause poor performance. As you may already know, collections include nested tables, VARRAYs, index-by tables, and host arrays.

Binding is the assignment of values to PL/SQL variables in SQL statements.
Bulk Binding

**Definition:**
- Bind entire collection/array of values at once, rather than loop to perform fetch, insert, update, and delete on multiple rows

**Keywords to support bulk binding:**
- **FORALL** instructs PL/SQL engine to bulk-bind input collections before sending them to the SQL engine
  
  ```sql
  FORALL index IN lower_bound..upper_bound
  sql_statement;
  ```

- **BULK COLLECT** instructs SQL engine to bulk-bind output collections before returning them to the PL/SQL engine
  
  ```sql
  ... BULK COLLECT INTO
  collection_name[,collection_name] ...
  ```

**Bulk Binding**

Bulk binding is binding an entire collection at once rather than iteratively. Without bulk binding, the elements in a collection are sent to the SQL engine individually, whereas bulk binds pass the entire collection back and forth between the two engines.

**Improved Performance**

Using bulk binding, you can improve performance by reducing the number of context switches required to run SQL statements that use collection elements. With bulk binding, entire collections, not just individual elements, are passed back and forth.

**Keywords to Support Bulk Binding**

Oracle8i PL/SQL supports new keywords to support bulk binding. These are:

**FORALL**

The keyword FORALL instructs the PL/SQL engine to bulk-bind input collections before sending them to the SQL engine. Although the FORALL statement contains an iteration scheme, it is not a FOR loop.

**BULK COLLECT**

The keywords BULK COLLECT instruct the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. This allows you to bind locations into which SQL can return retrieved values in bulk. Thus you can use these keywords in the SELECT INTO, FETCH INTO, and RETURNING INTO clauses.
Bulk Binds - Examples

The slide contains three examples:

1. The first example shows how to use FORALL with an INSERT statement.
2. The second example shows how to use BULK COLLECT INTO with a SELECT statement.
3. The third example shows how to use BULK COLLECT INTO with a DELETE statement.
Bulk Binds - Example

Example: FORALL ... BULK COLLECT INTO

```sql
FORALL j IN 1..1000
    DELETE Orders
    WHERE orderDate = oldDates(j) -- bulk
    AND orderLoc  = oldLoc      -- scalar
RETURNING orderId
BULK COLLECT INTO deletedIds;
```

Bulk Binds - FORALL .... BULK COLLECT INTO Example

In the example above, the FORALL statement causes the DELETE statement to be executed once for each different element of the oldDates table. Many rows can be deleted for each instance of the WHERE clause; all the orderIds returned by different executions of the DELETE statement are appended to (or bulk collected into) the deletedIds table.
When to Use Bulk Binds

Consider using bulk binds when you have:
• SQL statements inside PL/SQL loops
• A need for using collection elements as bind variables
• A need for processing four or more rows together in an iteration. The more rows affected by a SQL statement, the greater the gains.

Internal Oracle benchmark tests using bulk binds show performance improvements of up to 30%.
NOCOPY Parameter Passing Hint (Problem in Pre-8i Releases)

As we know, the IN OUT parameters in PL/SQL are passed using copy-in and copy-out semantics. The example demonstrates how the copy-in, copy-out semantics works. Let's assume that word_list corresponds to a dictionary and is a huge collection of (word, meaning) pairs.

This imposes huge CPU and memory overhead because the parameters need to copied-in when the routine is called and then copied-out when the routine returns. The overhead is HUGE when the parameters involved are large data structures, such as large strings or collections.
NOCOPY Parameter Passing Hint (Workaround in Pre-8i Releases)

This known performance problem forced the users to use package variables to “pass” arguments around.

Though this gets one around the performance problem imposed by by-value semantics, it has some obvious problems - the workaround increases the maintenance cost by making the programs non-modular, and results in higher maintenance costs.
NOCOPY Parameter Passing Hint

As you already know, PL/SQL supports three parameter passing modes - IN, OUT, and IN OUT.

The IN parameter is passed by reference. That is, a pointer to the IN actual parameter is passed to the corresponding formal parameter. So, both parameters reference the same memory location, which holds the value of the actual parameter. However, the OUT and IN OUT parameters are passed by value.

When an application passes parameters using the OUT and IN OUT modes, the database copies the parameter values to protect the original values in case the function or procedure raises exceptions. Passing large structures, such as strings and collections, places a burden on memory and CPU resources. One alternative is to use package-global variables, rather than parameters, but this results in non-modular programs.

Oracle8i supports a new NOCOPY modifier for OUT and IN OUT parameters, which is a hint to the compiler to indicate that the compiler should attempt to pass IN OUT and OUT parameters by reference when possible.

Because the actual parameters are passed by reference rather than copied into the memory for the corresponding formal parameters, NOCOPY provides significant benefits, especially when passing large structures as parameters. Oracle internal benchmark testing showed improvements of 30 to 200% improvements for medium-to-large PL/SQL tables passed as parameters.

Note that the NOCOPY modifier is neither required nor allowed with IN parameters; IN parameters are always passed by reference.

The syntax for this is very simple and is shown on the slide.
NOCOPY Parameter Passing Hint

Semantics:

- NOCOPY is a hint, not a directive; compiler attempts to pass by reference but does not promise to do so:
  - When the call is a remote-procedure call
  - When the actual parameter being passed is an expression value
  - When implicit conversion is involved while binding an actual to a formal parameter

NOCOPY Parameter Passing Hint

Semantics

Since NOCOPY is a hint and not a directive, Oracle does not guarantee that a parameter will be passed by reference, although an attempt is made to pass the actual by reference when possible. Here are few example cases where the parameter will NOT be passed by reference:

- when the call is a remote-procedure call
- when the actual parameter being passed is an expression value
- when there is an implicit conversion involved when binding an actual parameter to a formal

Because pass by reference is not guaranteed to happen, there are some semantics that are not guaranteed to happen:

- If you had true pass-by-reference and you pass the same actual parameter across different formal parameters, you would expect aliasing to happen. In the case of NOCOPY, there is no guarantee that aliasing will ALWAYS happen under these conditions. So you must not rely on aliasing behavior.

- If you had COPY IN / COPY OUT, you would expect that uncommitted changes would be rolled back if you had an unhandled exception. In the case of true pass-by-reference, you would expect that uncommitted changes would NOT be rolled back if you had an unhandled exception. However, in the case of NOCOPY, since it is not guaranteed to be pass-by-reference, there is no guarantee that uncommitted changes will NOT be rolled back upon an unhandled exception. So you must not rely on rollback-on-exception semantics when using NOCOPY.
Transparent Performance Improvements

- Optimization of package STANDARD builtins
- Faster anonymous block execution
- Faster RPC parameter passing
- Scalability (more users) improvements
- Caching of DLLs for improved external procedure performance

Transparent Performance Improvements

In addition to the enhancements discussed in this article, it’s worth noting that Oracle has also tuned the PL/SQL interpreter in several ways. For example, Oracle has optimized the built-in functions in package STANDARD for faster performance. In internal benchmark testing, calls made to built-ins--TO_CHAR, TO_DATE, and SUBSTR, for example--operated 10 to 30 percent faster in Oracle8i than they did in Oracle8 or Oracle7.
New Testing, Debugging, and Tuning Capabilities

The right information can help developers improve performance, trace flawed programming logic, and debug compilation and run-time errors. For PL/SQL developers, Oracle8i includes several enhancements and features that will help design, develop, and fine-tune applications.

To enable third-party vendors to build a strong set of debugging and analysis tools, Oracle8i includes the Probe family of APIs in PL/SQL in various Oracle-supplied packages, such as the DBMS_PROFILER package.

In this section, we discuss the Probe Family of APIs, and go into some details regarding the Debugger, Profiler, and Tracing tools.
Probe Family of APIs:

Even prior to Oracle8i, Oracle has provided an API for debugging through the DBMS_DEBUG package. This is the interface that is currently being used by our tools vendors.

Oracle supplies new packages DBMS_PROFILER and DBMS_TRACE with Oracle8i, which will allow you to profile and trace your code execution.

In this section, we discuss each of these interfaces. For more details, please refer to the Oracle8i Supplied Packages Reference, Release 8.1.5.
Tracing PL/SQL Execution

In large and complex PL/SQL applications, it can sometimes get difficult to keep track of subprogram calls when a number of them call each other. By tracing your PL/SQL code you can get a clearer idea of the paths and order in which your programs execute.

While a facility to trace your SQL code has been around for a while, Oracle now provides an API for tracing the execution of PL/SQL programs on the server. You can use the Trace API, implemented on the server as the DBMS_TRACE package, to trace the execution of programs by function or by exception.

DBMS_TRACE

DBMS_TRACE provides subprograms to start and stop PL/SQL tracing in a session. The trace data gets collected as the program executes, and it is written out to the Oracle Server trace file (8.1.5) or database tables (8.1.6).

A typical trace session involves:

- Enabling specific subprograms for trace data collection (optional)
- Starting the PL/SQL tracing session
  (DBMS_TRACE.SET_PLSQL_TRACE)
- Running the application which is to be traced
- Stopping the PL/SQL tracing session
  (DBMS_TRACE.CLEAR_PLSQL_TRACE)
Controlling the Trace

- In large applications, enable specific subprograms for trace
- Choose tracing levels:
  - Tracing calls:
    - Level 1: Trace all calls
    - Level 2: Trace calls to enabled subprograms only
  - Tracing exceptions:
    - Level 1: Trace all exceptions
    - Level 2: Trace exceptions raised in enabled subprograms only

Controlling the Trace

Profiling large applications may produce a huge amounts of data which can be difficult to manage. Before turning on the trace facility, you have the option to control the volume of data collected by enabling specific subprograms for trace data collection.

During the trace session, you can choose to trace all program calls and exceptions, or enabled program calls and exceptions raised in them. Choose the tracing levels shown above.

Note: For more information on this topic, refer to the Oracle product documentation.
PL/SQL Profiler

- Tool to **localize** and **identify** PL/SQL performance problems
  - Counts the number of times each line was executed
  - Determines how much time was spent on each line
- Information is stored in database tables, and can be accessed at any desired granularity
- The Profiler is implemented with **DBMS_PROFILE** package. APIs to record information:
  - START_PROFILER(run)
  - STOP_PROFILER
  - FLUSH_DATE

**Profiling**

Oracle8i PL/SQL provides a new tool called the Profiler which can be used to determine the execution time profile (or runtime behavior) of applications. The Profiler can be used to figure out which part of a particular application is running slowly. Such a tool is crucial in identifying performance bottlenecks. It can help you focus your efforts on improving the performance of only the relevant PL/SQL components, or, even better, the particular program segments where a lot of execution time is being spent.

The Profiler provides functions for gathering “profile” statistics, such as the total number of times each line was executed; time spent executing each line; and minimum and maximum duration spent on execution of a given line of code. For example, developers can generate profiling information for all named library units used in a single session. This information is stored in database tables that can then be later queried.

The Profiler is controlled by the DBMS_PROFILE package. APIs available to record information are:

- **START_PROFILER**: In order to get profiling information, you first need to start the Profiler by calling the DBMS_PROFILE.START_PROFILER procedure and associating a specific “run” comment with the profile run.
- **STOP_PROFILER**: To stop the profiling, you must call the DBMS_PROFILE.STOP_PROFILER procedure.
- **FLUSH_DATA**: You may call the DBMS_PROFILE.FLUSH_DATA procedure during the session to save incremental data and to free memory for allocated profiler data structures.
Debugger

- The Probe API, DBMS_DEBUG, is a PL/SQL API to the PL/SQL debugger layer
- Used by 3rd party vendors to build debuggers
- Oracle Procedure Builder provides an editor/compiler/debugger GUI interface

Debugger

The Probe API, DBMS_DEBUG, is a PL/SQL API to the PL/SQL debugger layer. It is currently used by 3rd party vendors to build debuggers for PL/SQL. The Oracle Procedure Builder product (which is packaged with the Oracle Developer Suite), which provides a PL/SQL edit/compile/debug development environment, uses an internal version of the same debugger APIs.
Building Web Applications

- **UTL_HTTP package**
  - Can be used to make HTTP callouts from PL/SQL and SQL
- **UTL_SMTP package**
  - Allows programs written in PL/SQL to communicate over the Internet using SMTP
- **UTL_TCP**
  - Allows simple TCP/IP-based communication between the server and the outside world
- **PL/SQL Server Pages (PSP)**
  - Scripting solution for server-side Web application development

Building Web Applications

The Oracle8i Server and the Oracle Application Server provide a lot of very useful functionality that make it very easy to publish dynamic content on the web using PL/SQL. For more details, refer to the Oracle OpenWorld99 presentation entitled “Web-enabling PL/SQL based Database Applications using PL/SQL”.

---

Page 46
Interoperability with C, Java

The Oracle8 Server, Release 8.0 provides the ability to call, from PL/SQL stored procedures, C external procedures executing in separate processes. With the Oracle8i release, developers can create stored procedures using both PL/SQL and Java. Each language has its own strengths, and since both are supported equally by the Oracle8i database server and interoperate easily, you can leverage both in the same application.

PL/SQL is best used for database-intensive applications, while Java is best used for component-based (Enterprise JavaBeans components or CORBA) applications. Moreover, because interoperability between the two languages is an important part of Oracle’s strategy, you can write very powerful applications that take advantage of the strengths of both languages. In this section, we examine how PL/SQL and Java interoperate in the server.

Java can call SQL and PL/SQL using SQLJ and JDBC. For more details about this, refer to the Oracle8i Java Stored Procedures Developer’s Guide, Release 8.1.5 (Part# A64686-01), the Oracle8i JDBC Developer’s Guide and Reference, Release 8.1.5 (Part# A64685-01), and the Oracle8i SQLJ Developer’s Guide and Reference, Release 8.1.5.

To allow SQL and PL/SQL to call Java stored procedures, the Java stored procedures must be published to SQL and PL/SQL using call specifications.
Summary

What Oracle8i PL/SQL offers:
- **Easier Application Development**: native dynamic SQL, package body size limit increased, purity simplification, ORA-6502, ...
- **Faster Applications**: bulk binds, NOCOPY, ...
- **Richer Application Development**: invoker’s rights, autonomous transactions...
- **Interoperable with Java**: call specifications, ...
- **Tools**: better debugging, profiling, tracing, ...
- **Internet Support**: PSP, UTL_HTTP enhancements, ...

Summary (What you can do with Oracle8i PL/SQL):

PL/SQL is a very powerful, yet simple language that allows you to quickly and easily write procedural logic for data access. Oracle8i PL/SQL provides powerful, new functionality for enhanced control of the security, transaction management, SQL execution, performance, and scalability of your language. Features like “bulk binds” and “NOCOPY” as well as the new profiler tool allow you to tune your application’s performance. Features like native dynamic SQL, invoker’s rights programs, and autonomous transactions, allow you to write richer and more powerful applications much more easily.

Interoperability with Java is an important part of the dual-language strategy regarding PL/SQL and Java. The OAS PL/SQL cartridge as well as the UTL_HTTP package allow you to write internet applications very easily. In summary, Oracle8i PL/SQL provides you with a great deal of capabilities to allow you to write superior, high-performance applications very easily.